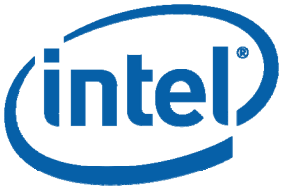


Intel® Thread Checker

Guide to Sample Code

Document Number: 313103-001US

Revision: 3.0

World Wide Web: http://www.intel.com

Disclaimer and Legal Information

Revision History

Document Number	Revision Number	Description	Revision Date
	1.0	Initial release.	2002
313103	3.0	• Content updated for 3.0 product version. • Apply new template.	May 2006

Contents

§

Figures

Tables

1 About this Document

This document presents examples of typical threading errors that can be detected by the Intel® Thread Checker such as *data races* and *deadlocks*. Separate but similar examples are provided for code on Windows* and on Linux* systems. The discussion in this document focuses on the Windows* version of the code, however, the analysis is similar.

The goals of this document and associated code samples are to help you learn to:

- Build code using the options required for Intel® Thread Checker.
- Use Thread Checker to diagnose potential issues in your threaded code.

1.1 Prerequisites

Before using this guide, you should know how to create and run an Activity using the Intel® Thread Checker Wizard and be familiar with basic product features as described in the *Intel® Thread Checker Getting Started Guide*. This guide and additional documentation are available from the **Start > Programs > Intel® Software Development Tools > Intel® Thread Checker 3.0 > Documentation and Support**.

1.2 Using this Document

To get the most of this document, you should:

1. Review the sample code.
2. Build the sample code using appropriate compiler options.
3. Create and run an Activity using the Intel® Thread Checker.
4. Review the diagnostics to understand the threading issues and possible solutions related to these code samples.

This *Guide to Sample Code* contains the following sections:

Table 1 Document Organization

Section	Description
Data Races	Finding a Common Data Race
HelloWorld	Finding a Subtle Date Race

Section	Description
Deadlock	Finding a Deadlock

1.3 Conventions and Symbols

The following conventions are used in this document.

Table 2 Conventions and Symbols used in this Document

`This type style`	Indicates an element of syntax, reserved word, keyword, filename, computer output, or part of a program example. The text appears in lowercase unless uppercase is significant.
This type style	Indicates the exact characters you type as input. Also used to highlight the elements of a graphical user interface such as buttons and menu names.
This type style	Indicates a placeholder for an identifier, an expression, a string, a symbol, or a value. Substitute one of these items for the placeholder.
[*items*]	Indicates that the items enclosed in brackets are optional.
{ *item* | *item* }	Indicates to select only one of the items listed between braces. A vertical bar (|) separates the items.
... (ellipses)	Indicates that you can repeat the preceding item.

1.4 Sample Code for Windows* Systems

By default, the Windows* version of the sample code is located in:
`C:\Program Files\Intel\VTune\tcheck\Samples`.
Each sample includes a Microsoft Visual Studio* workspace and project to enable the compilation and to simplify the build process of the code.

2 DataRaces Sample: Finding a Common Data Race

The code samples in this exercise demonstrate how to find *data races* (also known as *storage conflicts*) using the Intel® Thread Checker. Data races occur when multiple threads simultaneously access shared variables, which can corrupt data and give unexpected results. Because threads are scheduled in a non-deterministic manner, data races are not always apparent. The most common cause of data races is assuming a specific execution order between threads when in fact the execution order is uncertain.

Many factors can affect the order in which threads are executed, including processor speed, memory size, system load, and operating system version. Changes in any of these factors can hide errors or reveal new ones. These relationships make data race errors difficult to find.

The `DataRaces` sample code creates four threads, each of which increments a single shared variable. Use Intel® Thread Checker to find a typical type of data race in the code

The following steps describe how to build the Windows* version of the sample code and analyze it using Thread Checker.

2.1 Build the DataRaces Sample

Before you begin using Intel® Thread Checker, build `DataRaces` using one of the following methods. The sample code is located by default in `tcheck\Samples\DataRaces`.

2.1.1 Using the Intel® Compiler from the Command Line

To build using the Intel® Compiler from the command line, issue the command: `icl /Zi /Qtcheck DataRaces.c -o DataRaces.exe /link /fixed:no`

Optionally, use source instrumentation. Compile your code with the Intel compiler using the appropriate option.

On Windows* systems, use `/Qtcheck` and `/link /fixed:no`.

On Linux* systems, use -tcheck

Prepare your environment for command line data collection:

On Windows* systems, execute tcheckvars.bat.

On Linux* systems:

For sh or bash, execute ". tcheckvars.sh".

For C-shell or compatible systems such as tcsh, execute "source tcheckvars.csh".

In the command window, run:

tcheck_cl [<options>]<app_to_analyze>

where:

app_to_analyze is the name of the application you want to analyze

options are any of the available options.

2.1.2 Using Microsoft* Visual C++ 6.0

1. Open the `DataRaces.dsw` workspace.
2. Optionally, to use the Intel® Compiler, select **Tools > Intel(R) C++ Compiler Selection Tool** to change the compiler selection.
3. Use the **Build** menu to create the executable `DataRaces.exe`.

If you are not using the Intel compiler, you can ignore a warning about the `/Qtcheck` option being unknown. The `/Qtcheck` option is an Intel(R) compiler option that enables compile-time source instrumentation of the binary program.

2.1.3 Using Microsoft* Visual Studio or Visual C++ .Net Environment

1. Open the `DataRaces.dsw` workspace.

2. Click **Yes** to convert the workspace into a solution (`.sln`) file.
3. Optionally, to use the Intel® C++ Compiler, right-click on the project in the **Solution Explorer** and select **Convert to Intel(R) C++ Project System**.
4. Use the **Build** menu to create the executable `DataRaces.exe`.

If you are not using the Intel compiler, you can ignore a warning about the `/Qtcheck` option being unknown. The `/Qtcheck` option is an Intel(R) compiler option that enables compile-time source instrumentation of the binary program.

2.1.4 Build Options

To ensure best results with Thread Checker, the sample projects are set up to use certain build options. You need to use these options when building your own code for Thread Checker, so you may want to keep this list as a reference:

- **C/C++ > General** and **Linker**. Enable symbolic debugging information with the `/Zi` compiler and `/DEBUG` linker options. These options enable Thread Checker to provide you with symbolic information such as file name and line number along with its diagnostics. Thread Checker also supports other symbol formats generated by the Microsoft compilers including the `/ZI` and `/Z7` options.
- **Optimization**. Disable optimizations with the `/Od` compiler option.
  This setting is recommended to enable Thread Checker to accurately track symbolic information for your source code. This option is similar to using a debugger and ensures that line numbers are accurate. Turning on compiler optimizations is usually not beneficial for code run under Thread Checker.
- **C/C++ > Code generation > Runtime library**. Select thread-safe, multithreaded, debug runtime libraries with the `/MDd` compiler option.
  The compilers default to using non-thread safe run-time libraries (the `/MLd` compiler option) for use with single-threaded applications, so you must set this option when your application uses more than one thread.
- **C/C++ > Command Line > Additional Options**. Set the source instrumentation option with the `/Qtcheck` Intel® Compiler option.
  If you use a Microsoft* Compiler, this option is ignored with a warning.

CAUTION: When your application uses more than one thread, you must set the `/MDd` compiler option under the **C/C++ > Code generation > Runtime library** property. This option ensures that the thread-safe, multithreaded, debug runtime libraries are selected.

2.2 Collect Data

Create an Activity using Intel® Thread Checker to generate data for the `Debug\DataRaces.exe` image you compiled.

TIP: For a review of how to create an Activity using Thread Checker, see *Intel® Thread Checker Getting Started Guide* as described in *Prerequisites*.

TIP: If you need help completing the ***Intel® Thread Checker Wizard***, click the ***Help*** button on any page of the wizard for more details.

TIP: If Thread Checker does not show results, select ***Help*** > ***Search*** and type *"troubleshooting Thread Checker"* to search for relevant help topics. The help topic ***Troubleshooting Intel® Thread Checker*** offers possible solutions.

2.3 Analyze Results

Thread Checker offers a variety of features to help you identify and address threading issues in your sample code. Explore the following views to locate the data race in the `DataRaces.exe` sample.

2.3.1 Severity Distribution

The **Severity distribution** or **Graphical Summary View** in the right-hand pane of Thread Checker gives you a "big picture" snapshot of the number and types of errors found. The bars are color-coded to indicate the severity of problems found during the Activity run: red indicates the most severe errors, orange indicates warnings, and yellow indicates cautions. Blue and green indicate informational remarks.

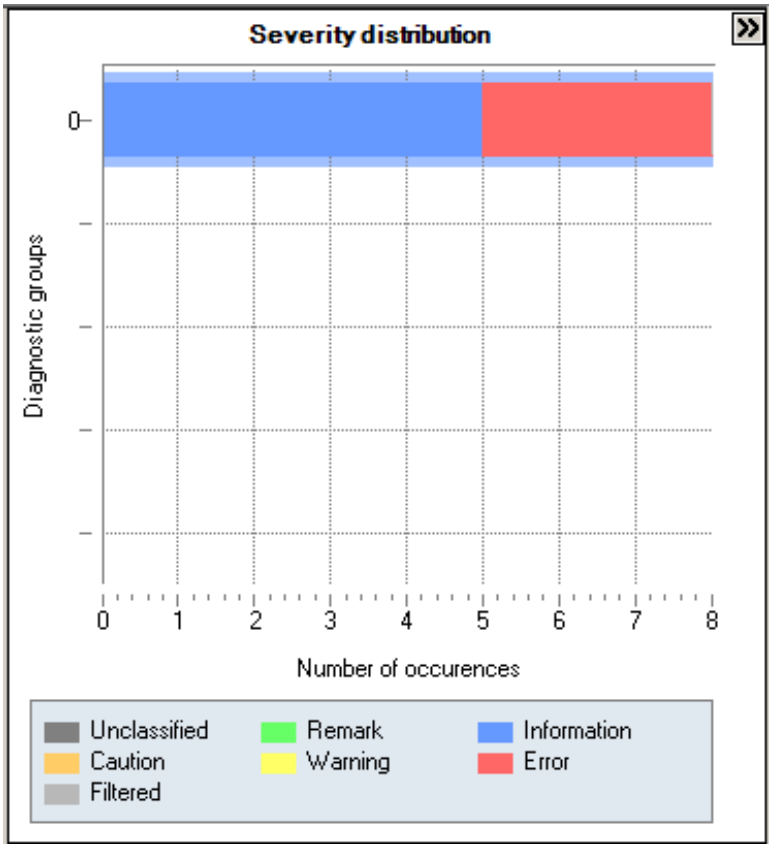


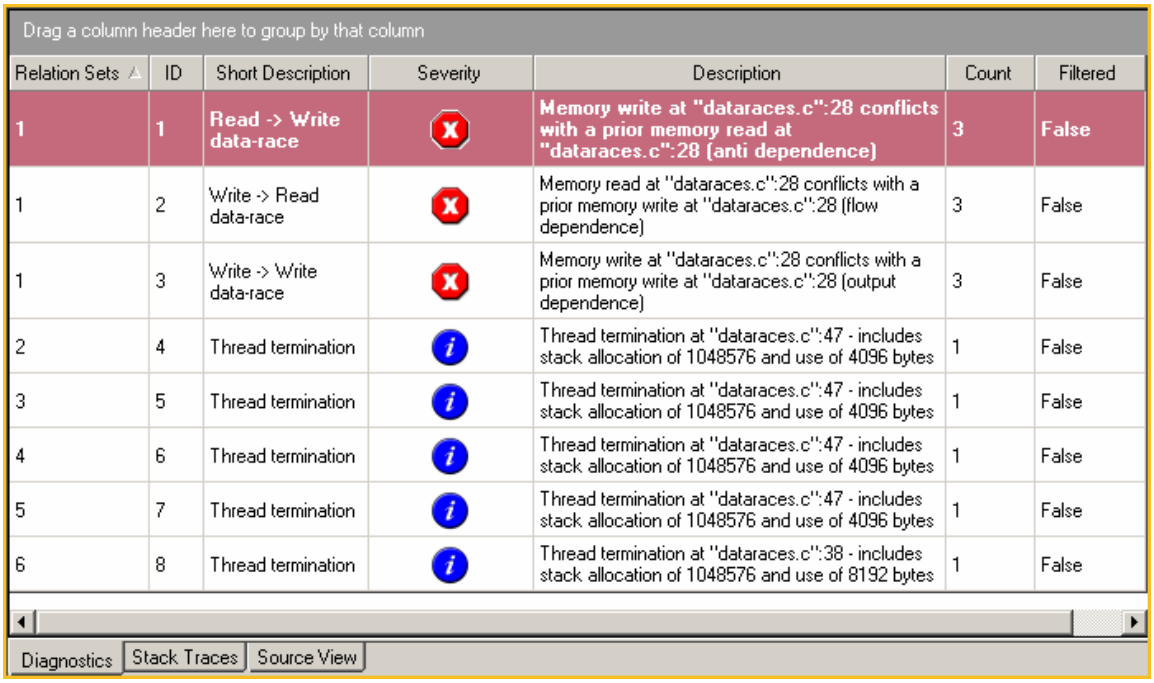
Figure 1: Severity distribution

Figure 1 shows the **Severity distribution** for the `DataRaces.exe` sample. Note that most of the diagnostics (five out of eight) are "blue" indicating that they are informational remarks. The remaining diagnostics are "red" indicating severe errors.

2.3.2 Diagnostics List

The **Diagnostics** list is Thread Checker's main view. It identifies each diagnostic by a unique **ID**, provides a **Short Description** and its estimated **Severity** as well as additional details including information about where the diagnostic occurred, its frequency, and more.

The following results were collected on code compiled using the Intel® Compiler, which supports the `/Qtcheck` option. Note that global variable name shows in the **Diagnostics** list.



Drag a column header here to group by that column

Relation Sets	ID	Short Description	Severity	Description	Count	Filtered
1	**1**	**Read -> Write data-race**		**Memory write at "dataraces.c":28 conflicts with a prior memory read at "dataraces.c":28 (anti dependence)**	**3**	**False**
1	2	Write -> Read data-race		Memory read at "dataraces.c":28 conflicts with a prior memory write at "dataraces.c":28 (flow dependence)	3	False
1	3	Write -> Write data-race		Memory write at "dataraces.c":28 conflicts with a prior memory write at "dataraces.c":28 (output dependence)	3	False
2	4	Thread termination		Thread termination at "dataraces.c":47 - includes stack allocation of 1048576 and use of 4096 bytes	1	False
3	5	Thread termination		Thread termination at "dataraces.c":47 - includes stack allocation of 1048576 and use of 4096 bytes	1	False
4	6	Thread termination		Thread termination at "dataraces.c":47 - includes stack allocation of 1048576 and use of 4096 bytes	1	False
5	7	Thread termination		Thread termination at "dataraces.c":47 - includes stack allocation of 1048576 and use of 4096 bytes	1	False
6	8	Thread termination		Thread termination at "dataraces.c":38 - includes stack allocation of 1048576 and use of 8192 bytes	1	False

Diagnostics | Stack Traces | Source View

Figure 2: Diagnostics list for DataRaces.exe sample

To focus on the most important diagnostics:

Group diagnostics. Drag column headers to the area above the table to group by different categories. For example, group by **Context[Best]** as shown in Figure 2 to get a high-level idea of where diagnostics occurred.

Filter diagnostics. Right-click and select **Filter diagnostic** to filter out the selected diagnostic. For example, right-click on the diagnostic with **ID 4** to filter out all "**Thread Info at "DataRaces.c".47...**" as shown in Figure 3.

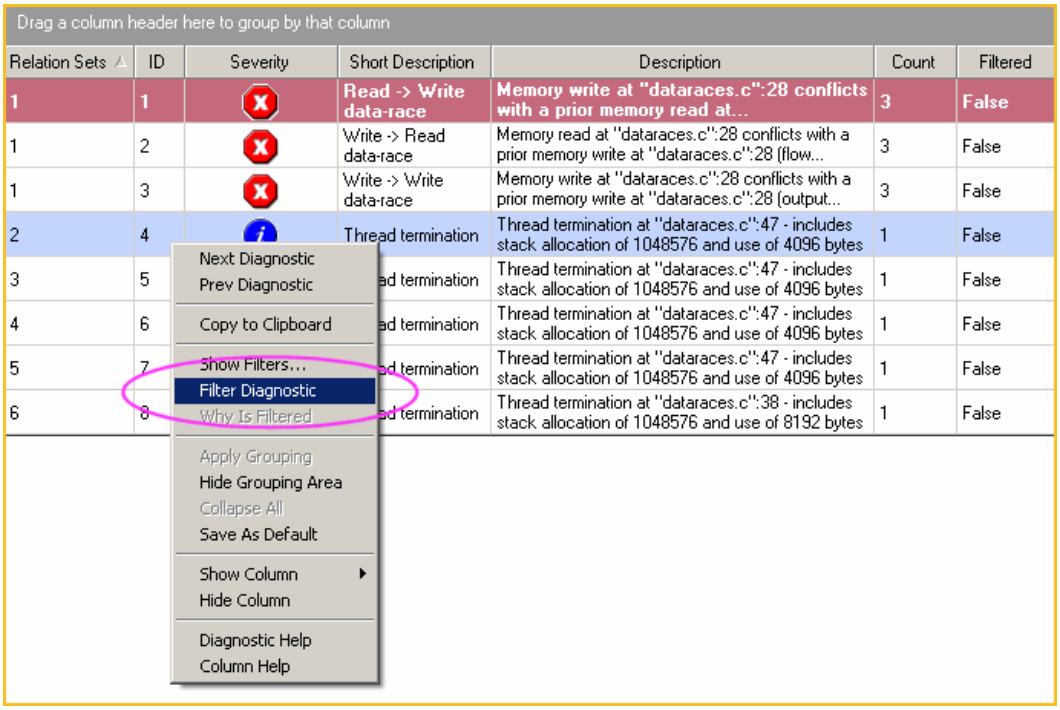


Figure 3: Filter Diagnostic

Show filters. Right-click and select **Show filters...** to modify, save, or append filters according to criteria you define as shown in Figure 4.

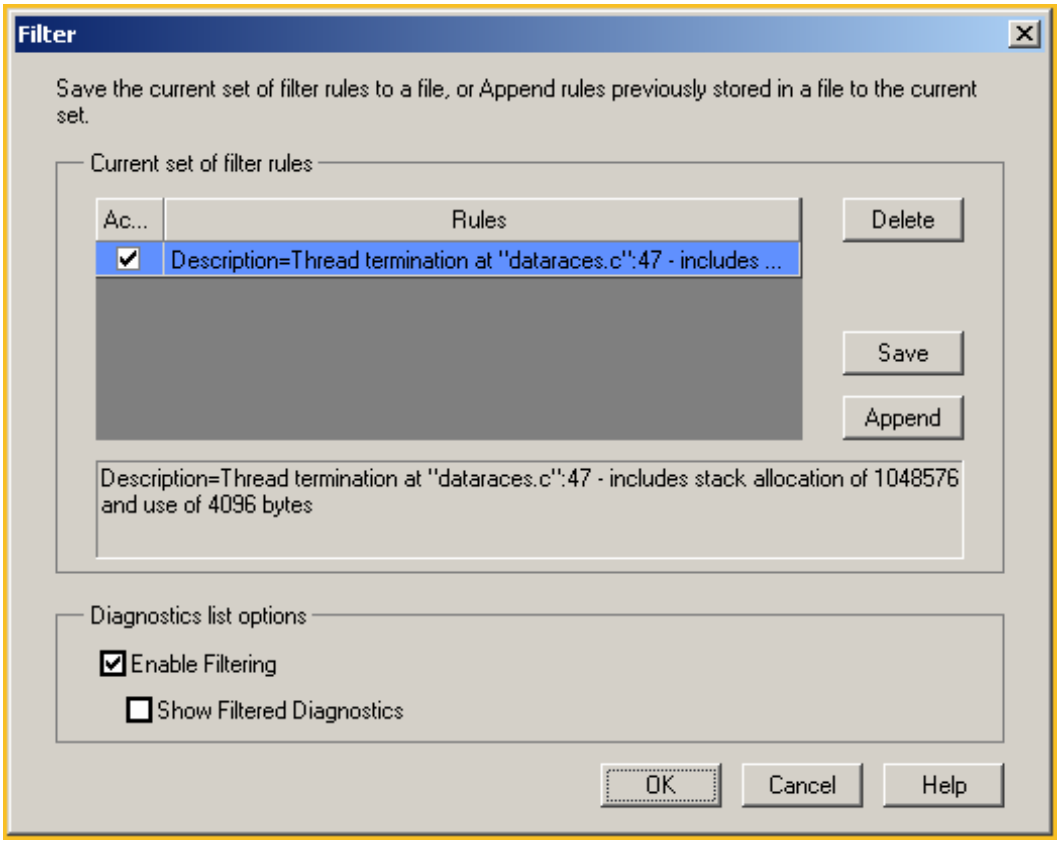


Figure 4: Filter dialog box

Show or hide columns to display. Right-click on a column and select **Hide Column** to remove it or select **Show Column >** and select from available columns to add a column.

Sort. Click a column heading to sort by that category. For example, click **Severity** to put most severe diagnostics at the top of your list.

2.3.3 Diagnostics Help

Most importantly, you use the **Diagnostics** view to understand issues that affect your threaded code. Right-click on a diagnostic and select **Diagnostic Help** to show a detailed description of the diagnostic, its possible causes and solutions.

For example, right-click on **ID 1** and select **Diagnostic Help** to learn more about **Read->write data races**.

2.3.4 Viewing the Source

Source View enables you to locate where a diagnostic occurred in your source code.

To examine the first error in the sample, in **Diagnostics** list, double-click the diagnostic with **ID 1** to select it and open the corresponding location in **Source View**.

Thread Checker reports multiple data races on the shared variable `globalX`. Looking over the code the error might seem impossible at first, because the increment statement appears to be protected by a `CRITICAL_SECTION`. In fact, the scope of the `CRITICAL_SECTION` variable `cs` is the problem. Each thread initializes a separate copy of the `CRITICAL_SECTION` so the threads are each locking a variable of local scope. Mutual exclusion can only be enforced if the `CRITICAL_SECTION` is visible to all threads attempting to enter a critical region.

To correct this error, move the declaration of the `CRITICAL_SECTION` to the same scope as the variable `globalX` and initialize the `CRITICAL_SECTION` in the `main()` function.

TIP: When Thread Checker does not know the symbol name of a variable, it may list it with a file name and source line if the application was compiled with the `/Od /Zi` options. It is listed as **Unknown** if no other information is available.

3 HelloWorld Sample: Finding a Subtle Date Race

Not all data races or storage conflicts can be solved using mutual exclusion objects. The next example contains a common, but difficult to diagnose, error. The program creates four threads, each of which prints a "`Hello, world from thread <thread_number>.`" message. Sometimes the program behaves as expected. Other times, however, duplicate numbers are printed.

3.1 Build the HelloWorld Sample

Build the `HelloWorld` sample, installed by default in `tcheck\Samples\HelloWorld` using Microsoft Visual Studio*.

3.2 Test Output

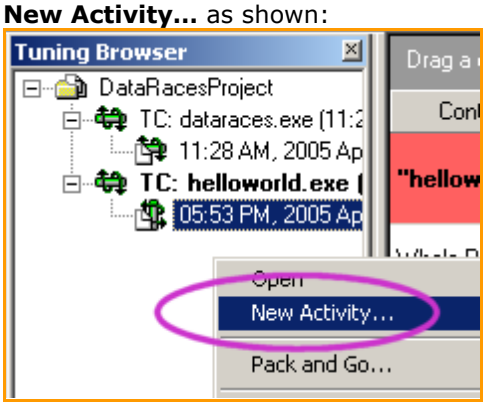
Run the executable several times to see if the output is correct. Notice that although you may see the correct output, there is still a subtle data race error in the code. This type of threading error is usually very difficult to detect because such errors do not always occur in every run of the software and are not immediately obvious.

3.3 Collect Data

In **Intel® Thread Checker**, do the following:

1. Create a new project as described in the previous example, or add a new Activity to your existing project by right-clicking in the **Tuning Browser** and selecting

New Activity... as shown:



2. Complete the **Intel® Thread Checker Wizard** for the `HelloWorld.exe` debug image you created.

Thread Checker creates an Activity and displays results.

3.4 Analyze Results

In this example, Thread Checker identified a single error, a **Write -> Read data-race** at **ID 1**, as shown in Figure 5:

Drag a column header here to group by that column

Relation Sets	ID	Short Description	Severity	Description	Count	Filtered
1	**1**	**Write -> Read data-race**		**Memory read at "helloworld.c":21 conflicts with a prior memory write at "helloworld.c":33 (flow dependence)**	**4**	**False**
2	2	Thread termination		Thread termination at "helloworld.c":35 - includes stack allocation of 1048576 and use of 4096 bytes	1	False
3	3	Thread termination		Thread termination at "helloworld.c":35 - includes stack allocation of 1048576 and use of 4096 bytes	1	False
4	4	Thread termination		Thread termination at "helloworld.c":35 - includes stack allocation of 1048576 and use of 4096 bytes	1	False
5	5	Thread termination		Thread termination at "helloworld.c":35 - includes stack allocation of 1048576 and use of 4096 bytes	1	False
6	6	Thread termination		Thread termination at "helloworld.c":28 - includes stack allocation of 1048576 and use of 8192 bytes	1	False

Diagnostics | Stack Traces | Source View

Figure 5: The Diagnostics for the HelloWorld.exe sample

To identify where this data race occurred in the code sample:

3. Double-click the diagnostic identified by **ID 1** to open the corresponding **Source View** location it.
4. As shown in Figure 6, **Source View** identifies a memory read line **21** is at conflict with a prior memory write at line **33**.

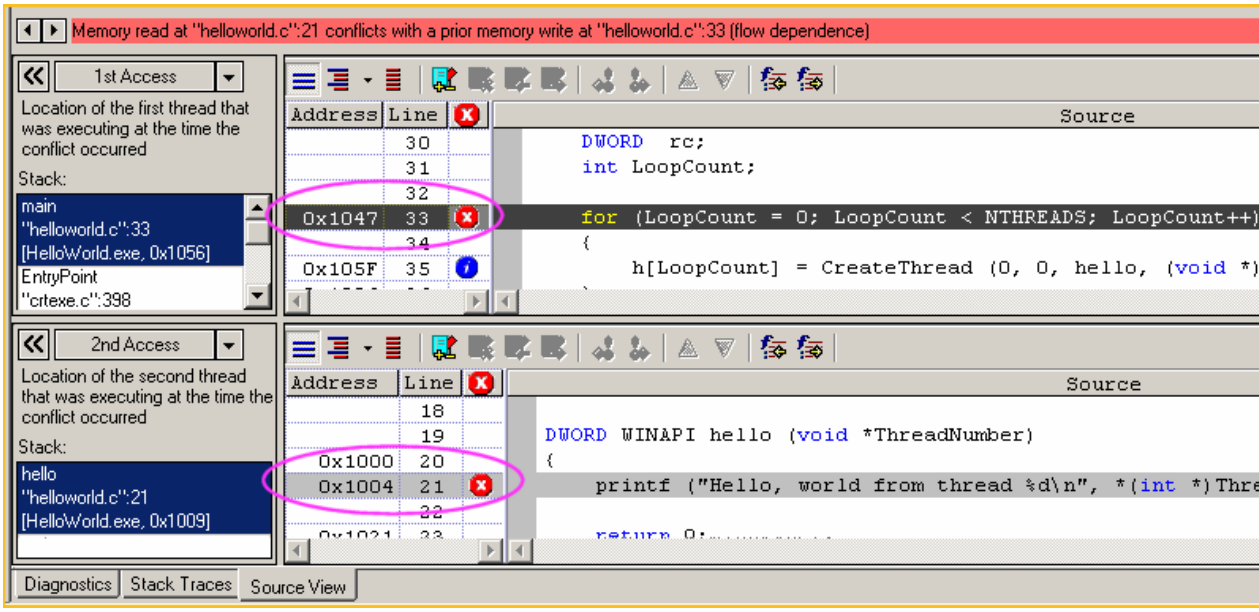


Figure 6: Source View for a data race

In this example, Intel® Thread Checker reports a **Write -> Read Data Race** (storage conflict) between variable `LoopCount` in `main()` and variable `ThreadNumber` in the `hello()` function. Specifically, variable `LoopCount` is being written by the parent thread while the child threads are dereferencing a pointer and reading it. All copies of `ThreadNumber` are pointers to the same address, the location of the loop iteration variable `LoopCount`, which is changing in `main()`. The code mistakenly requires that the `hello()` function begins executing immediately and completes before the next call to `CreateThread`.

The variable name `ThreadNumber` is listed in the diagnostic only if the executable is instrumented by the Intel® Compiler with `/Qtcheck`. Otherwise, Thread Checker shows `unknown` instead.

CAUTION: Never rely on a particular order of execution in your multithreaded code. Without explicit synchronization, threads can execute asynchronously with respect to other threads. This common error often causes multithreaded applications to exhibit non-deterministic behavior.

4 Deadlock Sample: Finding a Deadlock

This example shows you how to use Intel® Thread Checker to detect a deadlock and conditions that could lead to a deadlock. Deadlocks occur when a thread must wait for a resource that it can never acquire. Bad locking hierarchies are a common cause of deadlocks.

A mutex is a synchronization object used to allow multiple threads to serialize their access to a shared resource. The name derives from the capability it provides: *mutual exclusion.* The thread that locked a mutex becomes its owner and remains the owner until that same thread unlocks the mutex. The example program in this section uses the Windows* `CRITICAL_SECTION` variable to enforce mutual exclusion.

The following example, though contrived, illustrates how an incorrect locking hierarchy can cause deadlock. The two threads created in this program acquire two `CRITICAL_SECTION` variables in reverse order. If both threads obtain only the first `CRITICAL_SECTION`, deadlock results because the second `CRITICAL_SECTION` never becomes available. It is possible for one thread to acquire both `CRITICAL_SECTION` variables and avoid deadlock. However, multithreaded programs that depend on a particular order of execution are likely to fail.

4.1 Build the Deadlock Sample

Build the Deadlock sample located by default in `\tcheck\Samples\Deadlock` using the compiler of your choice.

4.2 Collect Data

In Thread Checker, do the following:

5. Create a new Activity: right-click in the **Tuning Browser** and select **New Activity...**.
6. Complete the **Intel® Thread Checker Wizard** for the `Deadlock.exe` debug image you created.

By default, Thread Checker forces termination of programs upon deadlock or program exit. You can change this setting under **Configure > Options > Intel® Thread**

Checker > Collector by deselecting the checkbox, **Forcefully terminate the program upon deadlock or program exit**.

If execution freezes, it may be due to *deadlock*. When deadlock occurs:

- Wait a few seconds, then press the stop button  .
- Close the console window to end the process. Thread Checker shows you the data it collects.

4.3 Analyze Results

The **Diagnostics** list shown in Figure 7 indicates several potential deadlocks, since synchronization objects were acquired in the wrong order. In addition to the **Read->Write** data race errors, Thread Checker diagnosed four deadlock errors indicating a potential deadlock due to bad locking hierarchy

Drag a column header here to group by that column

Relation Sets	ID	Short Description	Severity	Description	Count	Filtered
1	**1**	**Read -> Write data-race**		**Memory write at "deadlock.c":29 conflicts with a prior memory read at...**	**1**	**False**
1	2	Read -> Write data-race		Memory write at "deadlock.c":31 conflicts with a prior memory read at "deadlock.c":65 (anti...	1	False
2	3	A sync object was acquired in the wrong order		A synchronization object "deadlock.c":30 was acquired in the wrong order at "deadlock.c":28	1	False
2	4	A sync object was acquired in the wrong order		A synchronization object "deadlock.c":43 was acquired in the wrong order at "deadlock.c":41	1	False
3	5	Thread termination		Thread termination at "deadlock.c":63 - includes stack allocation of 1048576 and use of 4096 bytes	1	False
4	6	Thread termination		Thread termination at "deadlock.c":62 - includes stack allocation of 1048576 and use of 4096 bytes	1	False
5	7	Thread termination		Thread termination at "deadlock.c":53 - includes stack allocation of 1048576 and use of 8192 bytes	1	False

Diagnostics | Stack Traces | Source View

Figure 7: Diagnostic list for Deadlock.exe example

To find the source location of the first deadlock error, double-click the **deadlock error** at **ID 3** to select it and open the corresponding **Source View** location.

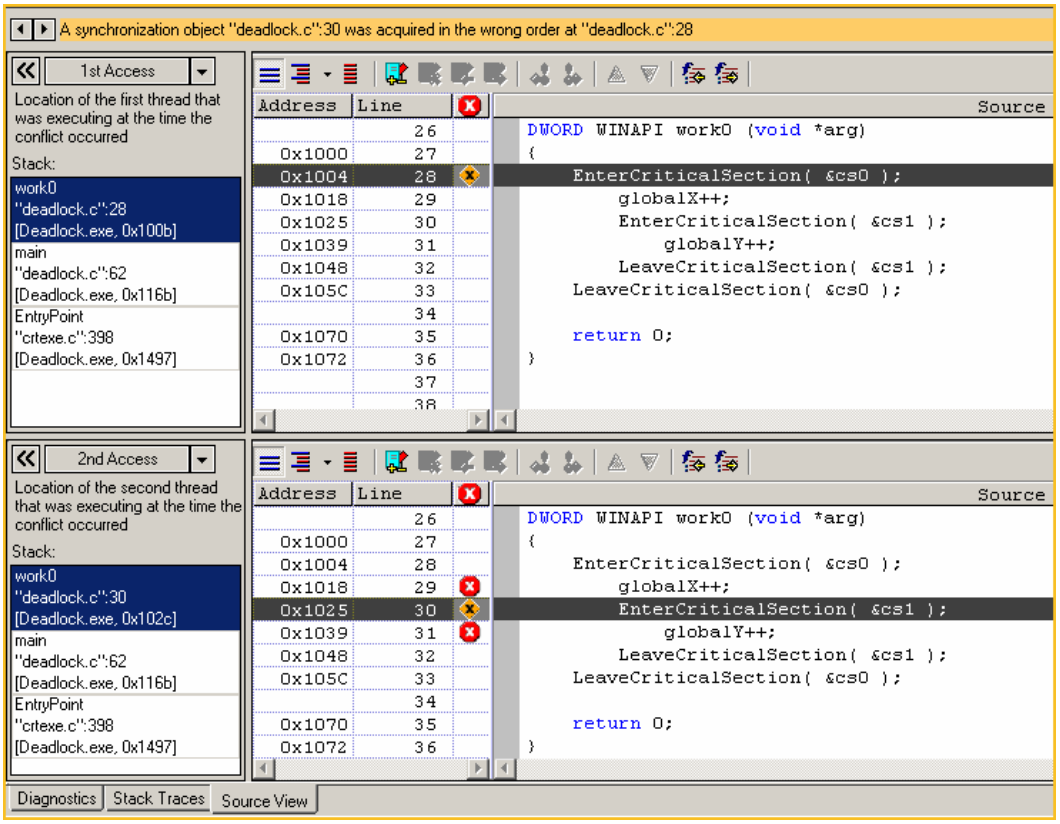


Figure 8: Source view showing the location of a potential deadlock

The **1st Access** and **2nd Access** panes of **Source View** show the location of a potential deadlock because of a locking hierarchy in function `work0`.
The easiest way to avoid deadlock when using locking hierarchies is to be sure to always lock and unlock the `CRITICAL_SECTION`s in the same order for all threads. So, in this example, both `work0()` and `work1()` should first call `EnterCriticalSection(&cs0)` then call `EnterCriticalSection(&cs1)`.
This means that both threads should call `LeaveCriticalSection(&cs1)` first and then call `LeaveCriticalSection(&cs0)`.

The **Read -> Write** data races occur because the variables being updated are not consistently protected.

In `work0()`, `CRITICAL_SECTION cs0` protects `globalX` while in `work1()`, `CRITICAL_SECTION cs1` protects that variable.

TIP: To avoid deadlocks when using locking hierarchies, always lock and unlock critical sections in the same order for all threads.